EECS 470 Lab 5 Assignment

Note:

- The lab should be completed individually
- The lab check off is due by **Friday, February 23rd**

1 Introduction

In this lab assignment you will be designing a generically-sized First-In First-Out circular buffer (a `FIFO` buffer). A hardware `FIFO` buffer is just like a software FIFO queue: the write operation does a push to the back (tail) of the queue, and a read returns and pops the front (head) of the queue. Circular buffers are a fundamental hardware component and used throughout industry and in your final project. You will be writing a parameterized version here that you can reference and extend for the final project.

2 FIFO Description

Our `FIFO` module has the following interface:

```
module FIFO #(
    parameter SIZE = 16,
    parameter WIDTH = 32,
    parameter ALERT_DEPTH = 3
) (
    input                    clock, reset,
    input                    wr_en,
    input                    rd_en,
    input        [WIDTH-1:0] wr_data,
    output logic             wr_valid,
    output logic             rd_valid,
    output logic [WIDTH-1:0] rd_data,
    output logic             almost_full,
    output logic             full
);
```

A FIFO uses a circular buffer internally with two pointers to its head and tail. These are pointers, so if your buffer can hold 32 entries, the pointers must be at least $log_2 32 = 5$ bits each (be careful with rounding if this is a non-power of two). The *head* pointer tracks the head of the queue, where you can read and pop from, and the *tail* pointer tracks the tail of the queue, where you write new data.

We use read and write enable bits on the input to say whether someone is reading or writing, but they don't know if this was successful until they see the matching valid bit in the output.

We need the valid bits since we can't *read* when the queue is empty and we can't *write* when the queue is full.

But we also need to keep track of whether we're empty or full.

A FIFO should start in an "empty" state with both `head` and `tail` at 0. So a FIFO is empty when `head == tail`, but it's also *full* when `head == tail`! This is a problem, and there are a few good ways to solve it:

1. (easiest) We can keep track of the number of entries in the queue with a simple counter. Then we're empty when `entries == 0` and full when `entries == SIZE`.

2. (easy) Make the FIFO one larger than your desired size, and use `head == tail` for empty and `head == tail + 1` for full (but be careful with overflow issues!).
3. (medium) We can use valid bits to say whether each entry in the queue is valid or not. Then we're empty when `ALL(valid_bits) == 0` and full when `ALL(valid_bits) == 1`
4. (hardest) We can keep using `head == tail`, but add extra logic to our full and empty indicator variables to track whether our previous state increased or decreased the number of entries.

The first is the easiest but a bit redundant and requires maintaining an extra counter. The second requires the least external tracking logic, but means you can never fully utilize the space in the FIFO. The third has high register storage/overhead, but is quicker. And the last has the lowest cost in bits, but is harder to reason about and get correct when including the `almost_full` signal.

You may implement any of these strategies in your FIFO module for this lab, or something else if you feel so inclined.

We also add an extra complication of an "alert depth", which we use to tell other modules when we're almost full. This is useful in superscalar processors, which many of you will choose for your final project.

3 Assignment

Implement the `FIFO` module with the following functionality:

1. Maintain a set of `SIZE` (default of 16) `WIDTH`-bit memory elements, which should only be updated on the positive edge of `clock`
2. When `reset` is high on the rising clock edge, the buffer should become empty (i.e. an immediately following read request would not be valid).
3. If `wr_en` is high at the positive edge of the clock, `FIFO` should store the value of `wr_data` to the first free index in the FIFO (you will need to maintain some sort of tail pointer for this). If there is no more space (`SIZE` values have been written and not read), `FIFO` should not modify its state and set `wr_valid` to 0; otherwise, `wr_valid` should be 1.
4. If `rd_en` is high, `FIFO` should write the oldest unread value written to it on the `rd_data` output and increment its head pointer so that the next read will output the next oldest value. If there are no values currently saved, it should set `rd_valid` to 0; otherwise, `rd_valid` should be 1.
5. The output `full` should be 1 iff there is no more space in in the buffer. This indicates to external modules that they cannot write anything else to the buffer.
6. There is an additional output `almost_full` which should be asserted when there are exactly `ALERT_DEPTH` spaces remaining in the FIFO. This gives a warning to external modules that there is limited space remaining.
7. If you simultaneously read and write to an empty FIFO, then the read should be invalid because there is nothing to read. In other words, there is a minimum one cycle latency between data going in and out.
8. If you simultaneously read and write to a full FIFO, then the write should be valid because there it can write to the newly opened spot.

When running '`make`', set the `SIZE` by doing '`make <target> SIZE=i`' at the shell; replacing '`i`' with your desired size and '`<target>`' with whatever make command you wish to run. e.g. to run simulation with size 48, do '`make sim SIZE=48`'. Similarly, to adjust the bit-width of the FIFO entries, use '`make target WIDTH=i`' where '`i`' is the number of bits in the `rd` and `wr` lines. You can also adjust the `ALERT_DEPTH` parameter from the command line in the same way.

The Makefile communicates these variables to the synthesis script and synthesizes separate .vg files for each fifo size to reduce compile time. See the Makefile's new `Modules with Parameters` section for details.

You might also need to add the `-B` flag (`make sim -B SIZE=i`) to tell make that it needs to recompile everything when you change the size. And the `--assume-old=` flag can avoid re-synthesizing the .vg modules when using `-B`.

Your design should function correctly for any values of `SIZE`, `WIDTH`, and `ALERT_DEPTH` (including `ALERT_DEPTH`=0). It is up to you how to implement this functionality. We recommend using a read and write pointer to keep track of where you should read from and write to next in the buffer.

Your design should pass the testbench after synthesis as well. However you don't need to worry about clock period for this lab, just make a design that works.

Note, the testbench uses a separate file with SystemVerilog assertions to check for proper functionality. This is provided as an example in case you wish to incorporate similar assertions in your design. The `bind` command in the testbench binds this assertion module with the DUT so we don't have to code the assertions in the FIFO module. This makes for cleaner and more reusable code.

4 Tips

- Be careful how you deal with the head and tail pointers wrapping around to the start of the buffer.
- Use the Modulo (`%`) operator when updating the head and tail.
- The `clog2()` function is helpful for finding the number of bits required to encode a number. To make sure it works with powers of 2, you should add 1 to the argument (i.e.: `clog2(NUM+1)`)
- Think about what approach you want to use to differentiate between the buffer being empty or full. Each approach comes with varying degrees of difficulty and efficiency.
- Supporting only one read and write per cycle means we are using dual ported RAM. It is dual ported because there is one read port and one write port. It is possible to build FIFOs that support multiple reads and writes per cycle, but these are much more expensive since they will use SRAM instead.

5 Bonus Features

- Add the constraint that `rd_data` must be 0 when `rd_en` is 0. Can you add an assertion in `FIFO_sva.svh` to check this?
- What if you want to scan the FIFO to see if a value exists within it? This requires implementing a CAM (content addressable memory) which involves a for loop with a break statement.
- Define a struct (or copy one from project 3), and make the FIFO store that instead of the base logic type. Is there a way to do that without changing the code within the module at all? Note: thinking about this may be helpful for your final project

6 Submission

A script has been provided (`check.sh`) that compiles and runs the testbench for a few different test `SIZE`s, `WIDTH`s, and `ALERT_DEPTH`s. Once you are passing all of those cases, add yourself to the help queue and an instructor will check you off. Please be ready to show the script output and your `FIFO.sv` file.

Please also be ready to discuss how FIFOs are used in the final project.